StarShow当期02：Pipeline执行引擎涉及的开发问题-冉攀峰

分享动机

- 涉及Pipeline执行引擎的调研, 设计, 代码解析, 测试报告文档很多;
- 其他团队的同学已经掌握了Pipeline通识原理, 我们再次回顾一下Pipeline的核心机制;
- 本文主要给出一些涉及Pipeline框架的feature开发的注意事项.

基本概念

MPP调度/Pipeline执行引擎/向量化

MPP调度, Pipeline调度和向量化执行是StarRocks执行引擎的三驾马车, 三者正交.

- MPP调度: 多机scale, 多机并行加速.
- Pipeline调度: 单机多核scale, 多核并行加速.
- 向量化执行: 挖掘现代处理器单核计算潜力: SMID，cache-friendly, tight-loop的auto-vectorization, prefetch, multiple issue等等.

Pipeline的任务模型是: cooperative multitasking, 在其他编程领域, 同质异名的说法也比较多, 比如coroutine, call/cc, yield等等.

Fragment Instance并行 v.s. Pipeline并行

Fragment Instance并行

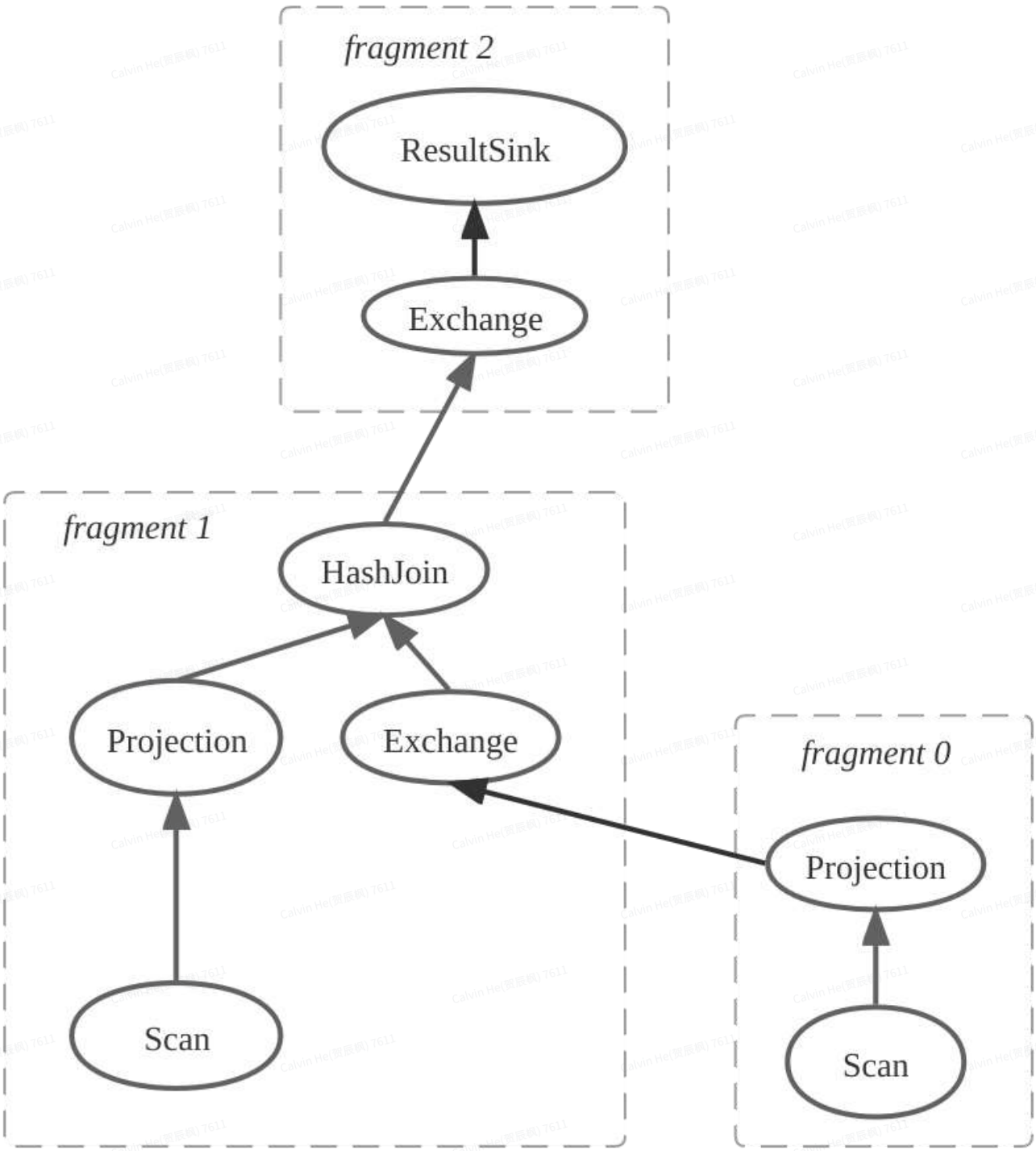
- Non-pipeline和Pipeline执行引擎都支持.
- 一个PlanFragment在一台BE上实例化多个Fragment Instance, 可以并发执行.调整并行度(parallel_exchange_instance_num/parallel_fragment_exec_instance_num), 意味着调整instance的数量.
- 在Non-pipeline引擎中, 实现比较简单, 任务队列+线程池模型, 本质上等价于Java ThreadPoolExecutor.

以Non-pipeline执行引擎举例

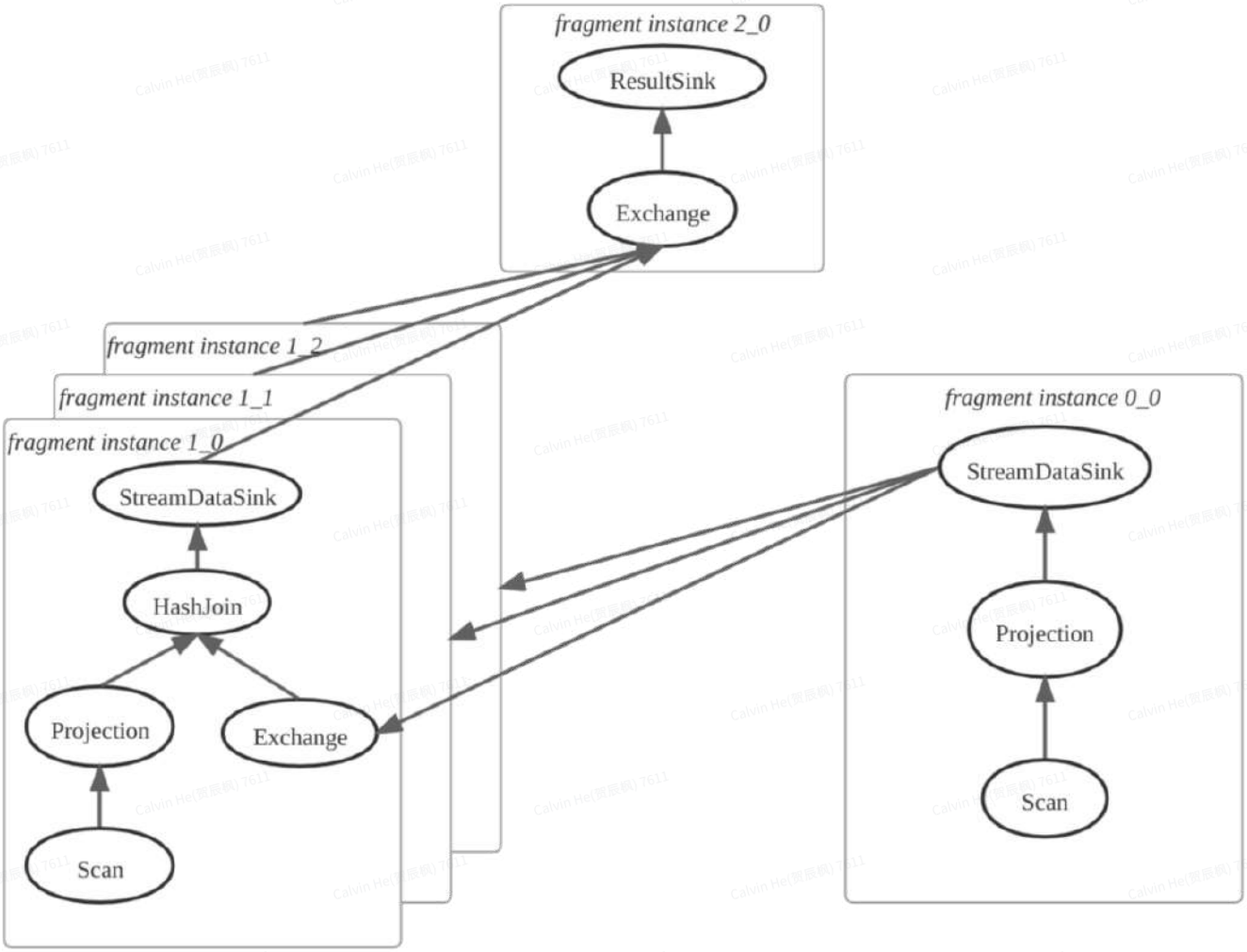
- SQL

```
1 select A.c0, B.c1 from A, B where A.c0  = B.c0
```

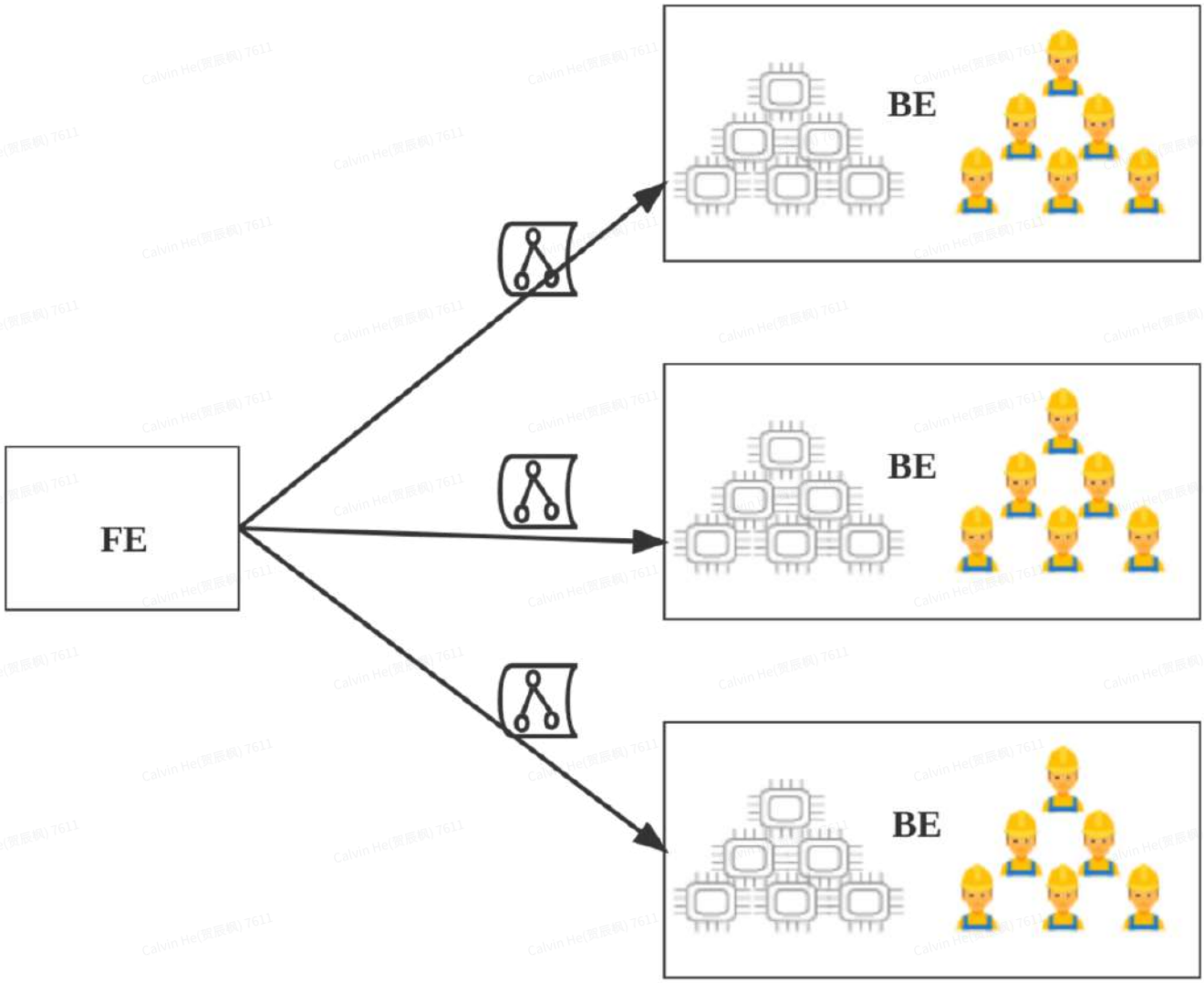
- PlanFragment



- Fragment Instance



- Fragment Instance execution

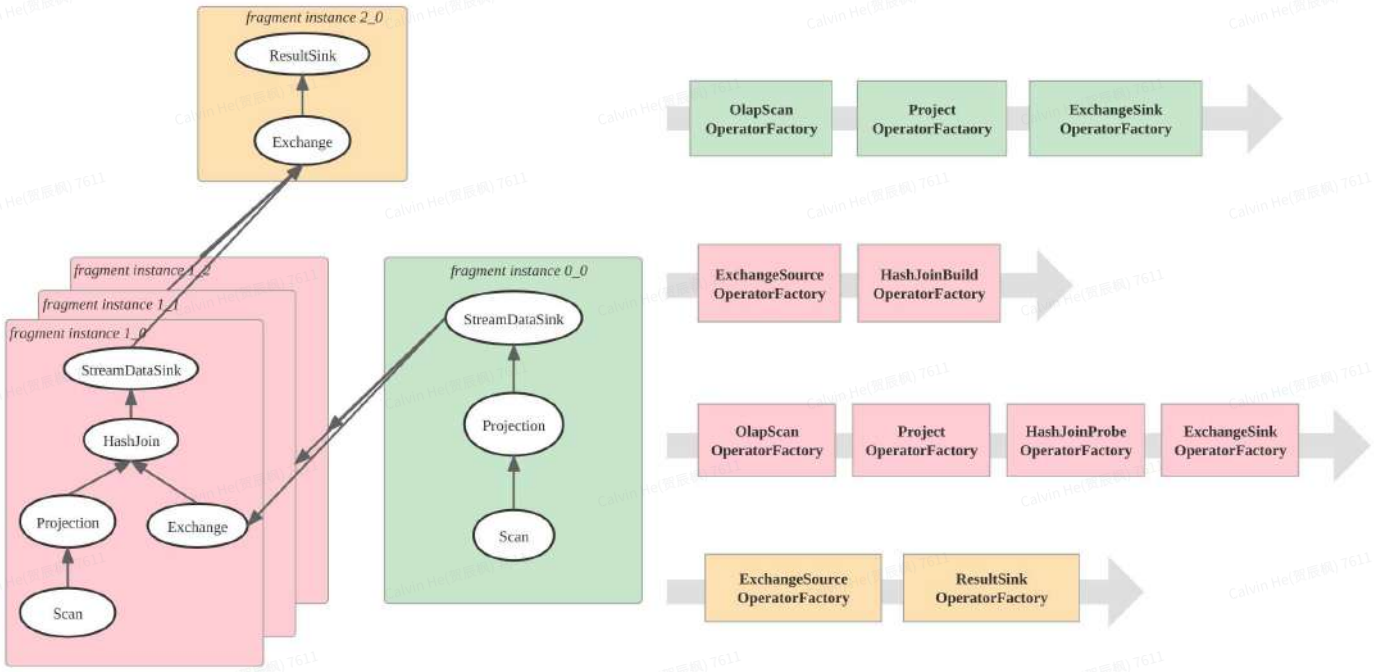


Pipeline并行

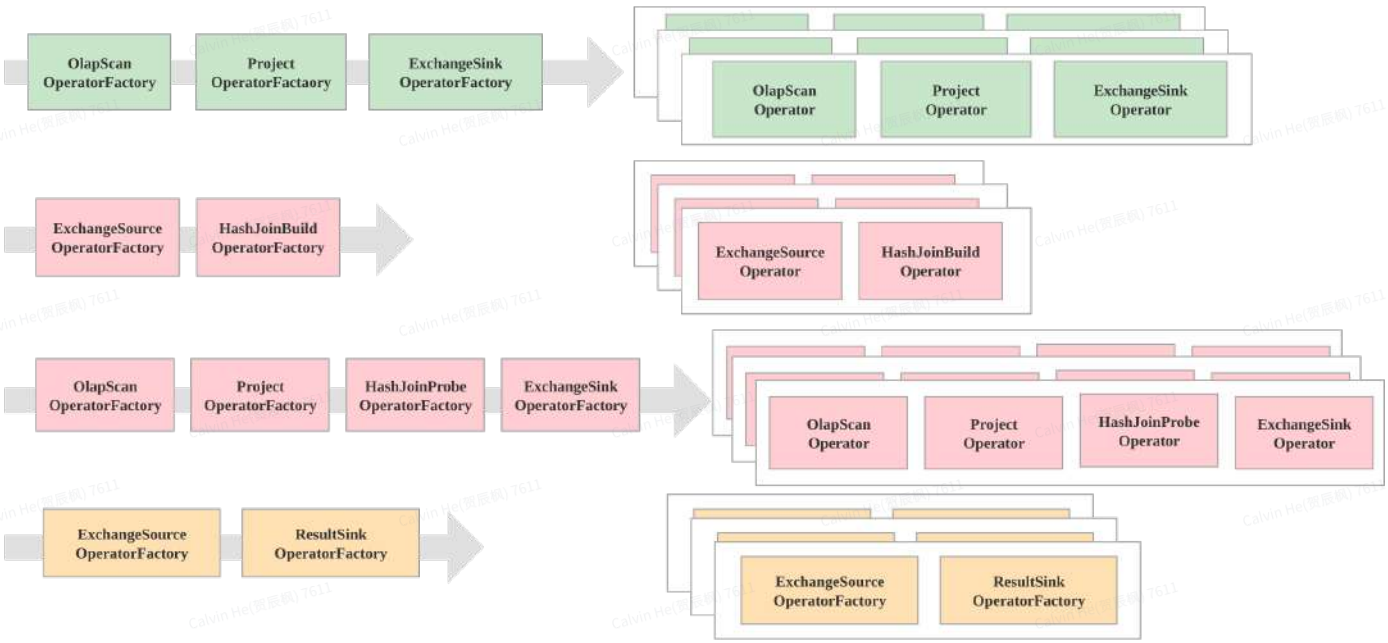
- 通常PlanFragment在每个目标BE上只实例化一个Fragment Instance.
- Fragment Instance拆分成若干Pipeline.
- Pipeline实例化成多个PipelineDriver.
- 采用固定数量的线程做执行线程, 用户可配, 默认为机器硬件线程数.
- PipelineDriver是用户态restartable task, PipelineDriver阻塞时不陷入内核挂起执行线程, 而是主动yield执行线程让其他就绪PipelineDriver执行, 一个PipelineDriver通常切入切出多次, 经过若干轮调度, 才能完成.
- Pipeline并行度的调整, 本质上改变PipelineDriver的数量.

举个例子

- Pipeline decomposition



- PipelineDriver instantiation



- Pipeline执行后面章节专门讲述

Pipeline基本概念

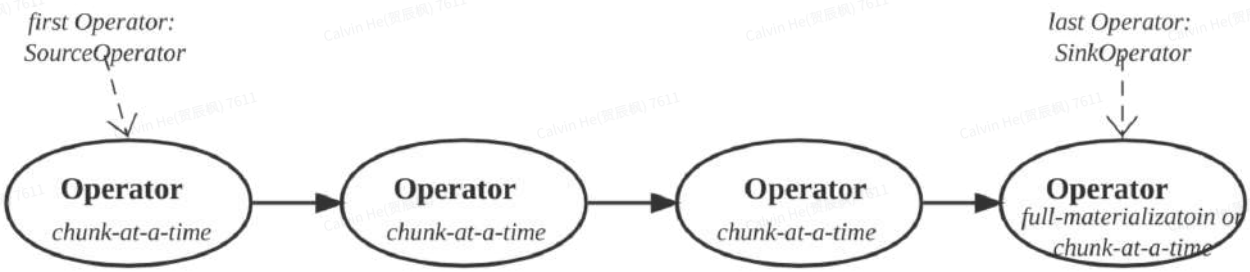
Pipeline

我们将算子的工作方式分成两类:

1. Chunk-at-a-time: 可以一个一个地处理chunk, 流式地向下游输出结果.
2. Full-materialization: 收到所有的数据后, 才能产生最终结果.

Pipeline是一组算子构成的链, 除Pipeline末尾算子外, 其他算子都以chunk-at-a-time方式工作.

- 开始算子为SourceOperator, 只有一个输出端.
- 末尾算子为SinkOperator, 只有输入端.
- Pipeline中间的算子只有一个输入端和输出端.



SourceOperator作为Pipeline的起始算子, 为Pipeline后续算子产生数据, SourceOperator获取数据的途经有:

1. 读本地文件或者外部数据源, 比如OlapScanOperator, ConnectorScanOperator;
2. 获得上游Fragment Instance的输出数据, 比如ExchangeSourceOperator;
3. 获得上游Pipeline的SinkOperator的计算结果, 比如LocalExchangeSourceOperator;

SinkOperator作为Pipeline的末尾算子, 吸收Pipeline的计算结果, 并输出数据, 输出途经有:

1. 把计算结果输出到磁盘或者外部数据源;
2. 把结果发给下游Fragment Instance, 比如ExchangeSinkOperator;
3. 把结果发给下游Pipeline的SourceOperator, 比如LocalExchangeSinkOperator;

Pipeline的中间算子, 即可获得前驱算子的输入, 又可以输出数据给后继算子.

Pipeline计算时, 从前向后, 先从SourceOperator获得chunk, 输出给下一个算子, 该算子处理chunk, 产生输出chunk, 然后推给再下一个算子, 这样不断地向前处理, 最终结果会输出到SinkOperator.对于每对相邻的算子, Pipeline执行线程调用前一个算子pull_chunk函数获得chunk, 调用后一个算子的push_chunk函数将chunk推给它. Pipeline执行, 直觉上, 和下面图片中的儿童游戏比较类似.

Pipeline拆分

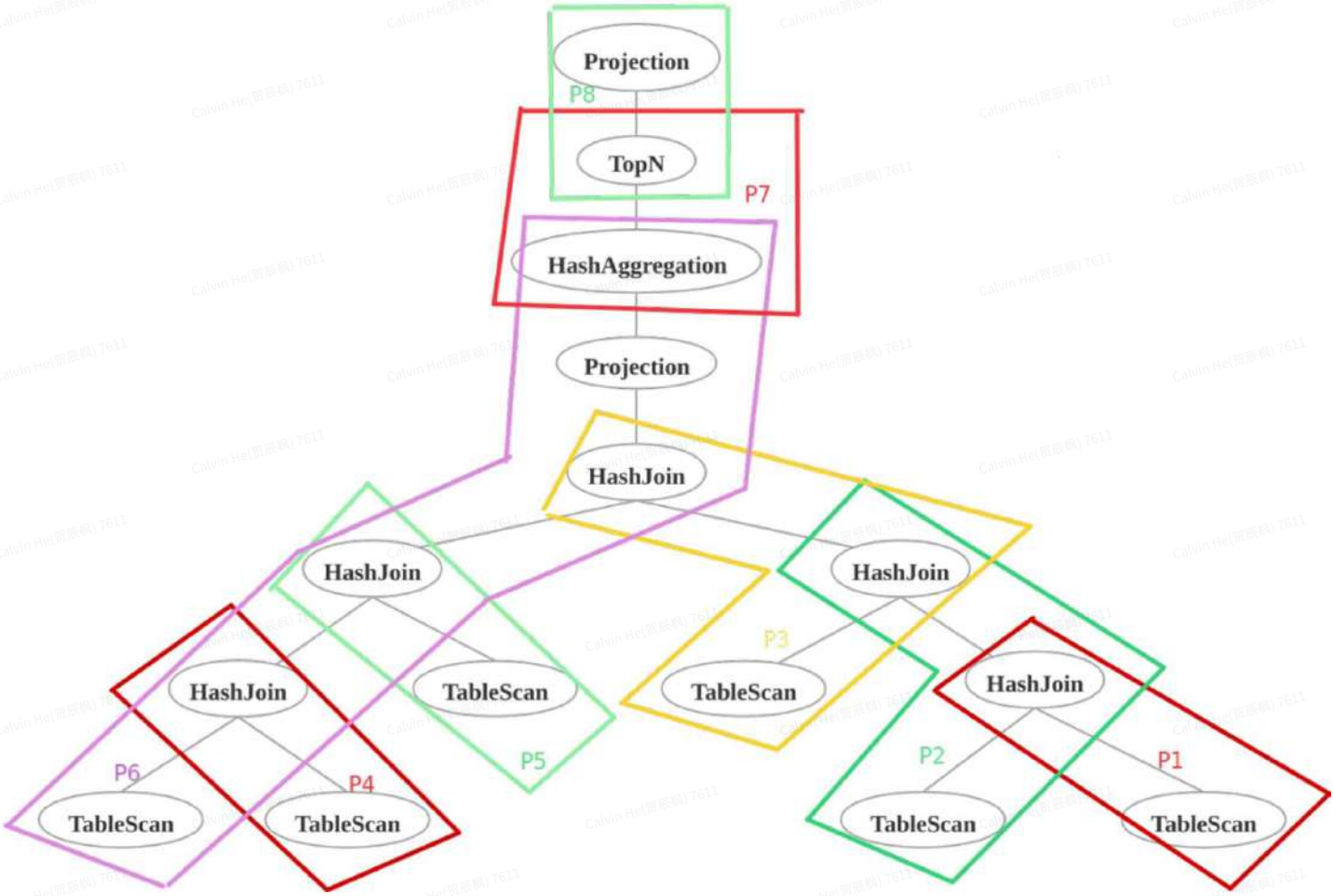
拆分标准

PlanFragment是树状的, 需要进一步转换为Pipeline, 这项转换工作由BE上的PipelineBuilder完成, FE对Pipeline的介入比较少. 一个PlanFragment可以拆分成若干条Pipeline, 相应地, PlanFragment中的物理算子也需要转换为Pipeline算子, 比如物理算子HashJoinNode需要转换为HashJoinBuildOperator和HashJoinProbeOperator. 拆分标准如下:

- 多输入/多输出算子(ExecNode)拆分成单输入单输出的Pipeline算子.
- full-materialized算子拆分一对sink/source算子, 分别位于上下游的pipeline中, sink和source算子通过共享的Context传递数据或者控制信息.
- DataSink转换为SinkOperator.
- 插入Local Exchange算子做并行度调整.

TPCH-Q5举例

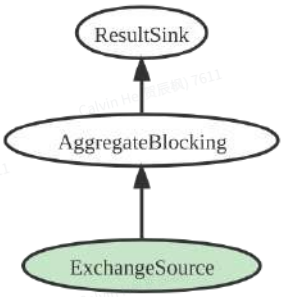
以TPCH-Q5为例，执行计划，可以划分成若干条Pipeline，Pipeline之间也存在上下游数据依赖. 如下图所示：



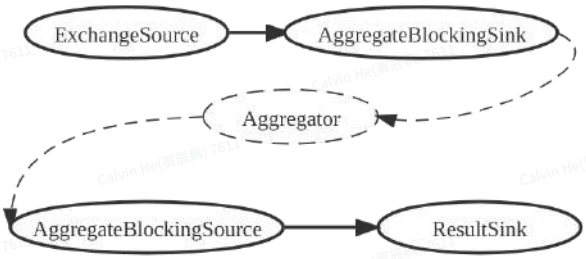
- P2 依赖 P1
- P3 依赖 P2
- P6 依赖 P3, P4, P5
- P7 依赖 P6
- P8 依赖 P7

复杂算子的拆分举例

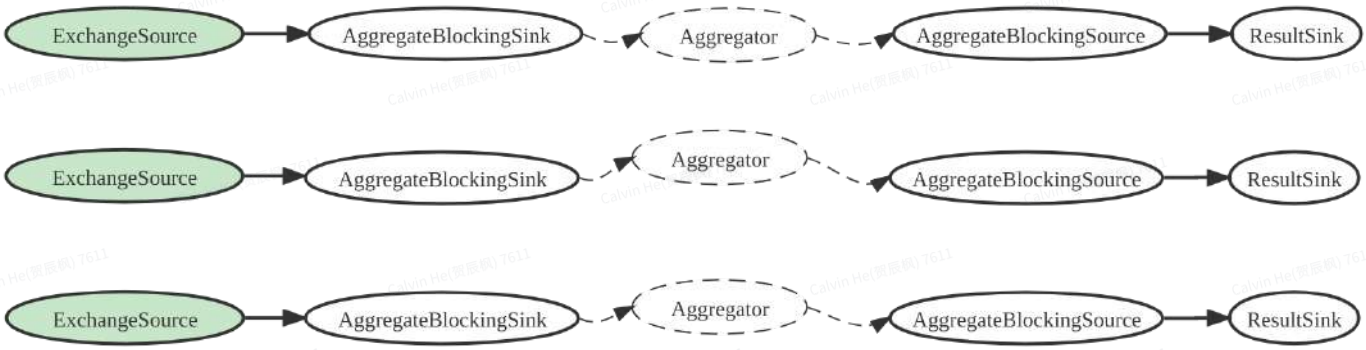
Agg算子



non-pipeline(fragment instance)

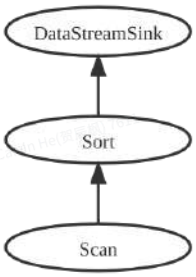


pipeline(DOP=1)



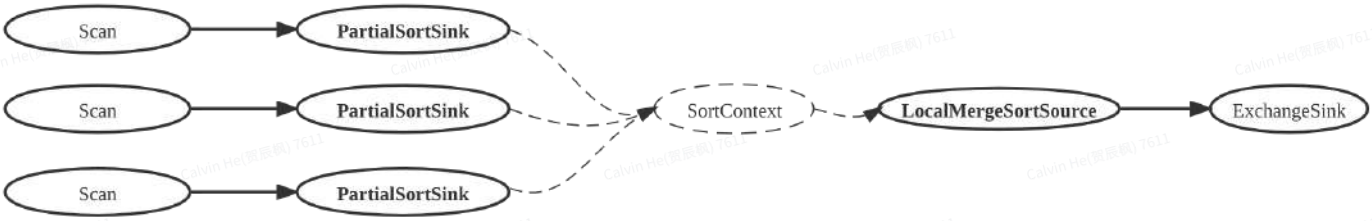
pipeline(DOP=3)

Sort算子



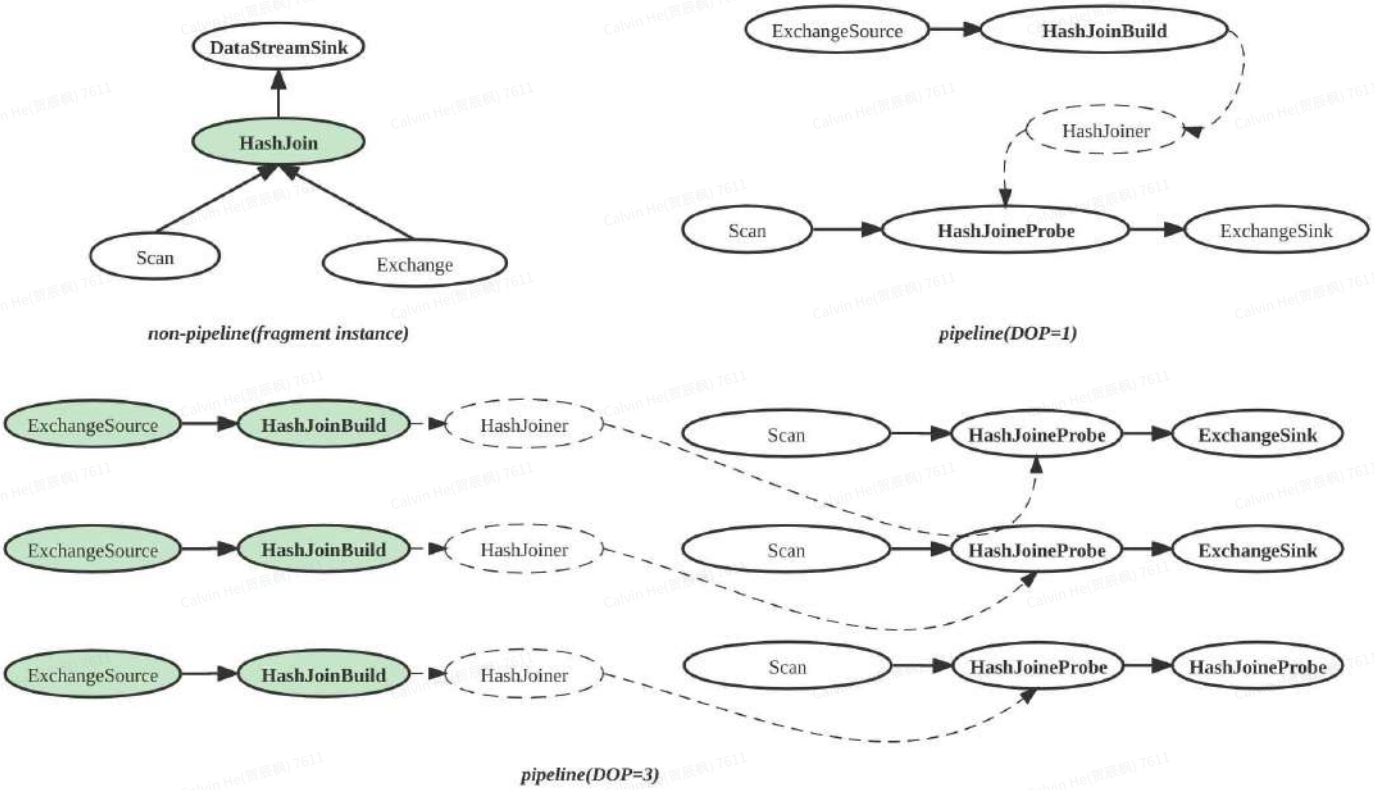
non-pipeline(fragment instance)

pipeline(DOP=1)



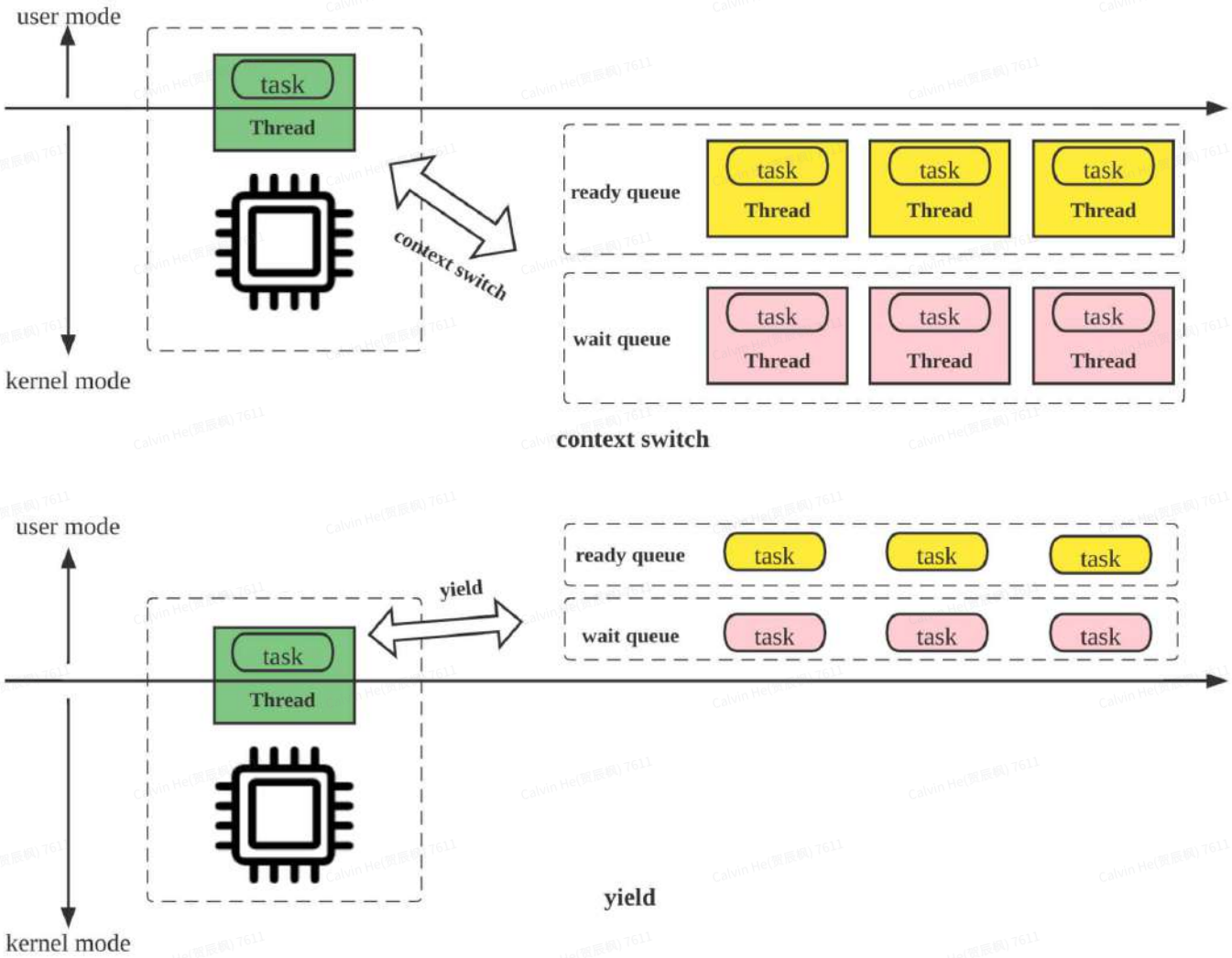
pipeline(DOP=3)

HashJoin算子



Pipeline调度

Cooperative multitasking v.s. multithreading



- Multitheading: 执行线程数量随着fragment instance数量而scale
- Multitasking: 执行线程数量固定
- 更低的调度开销.

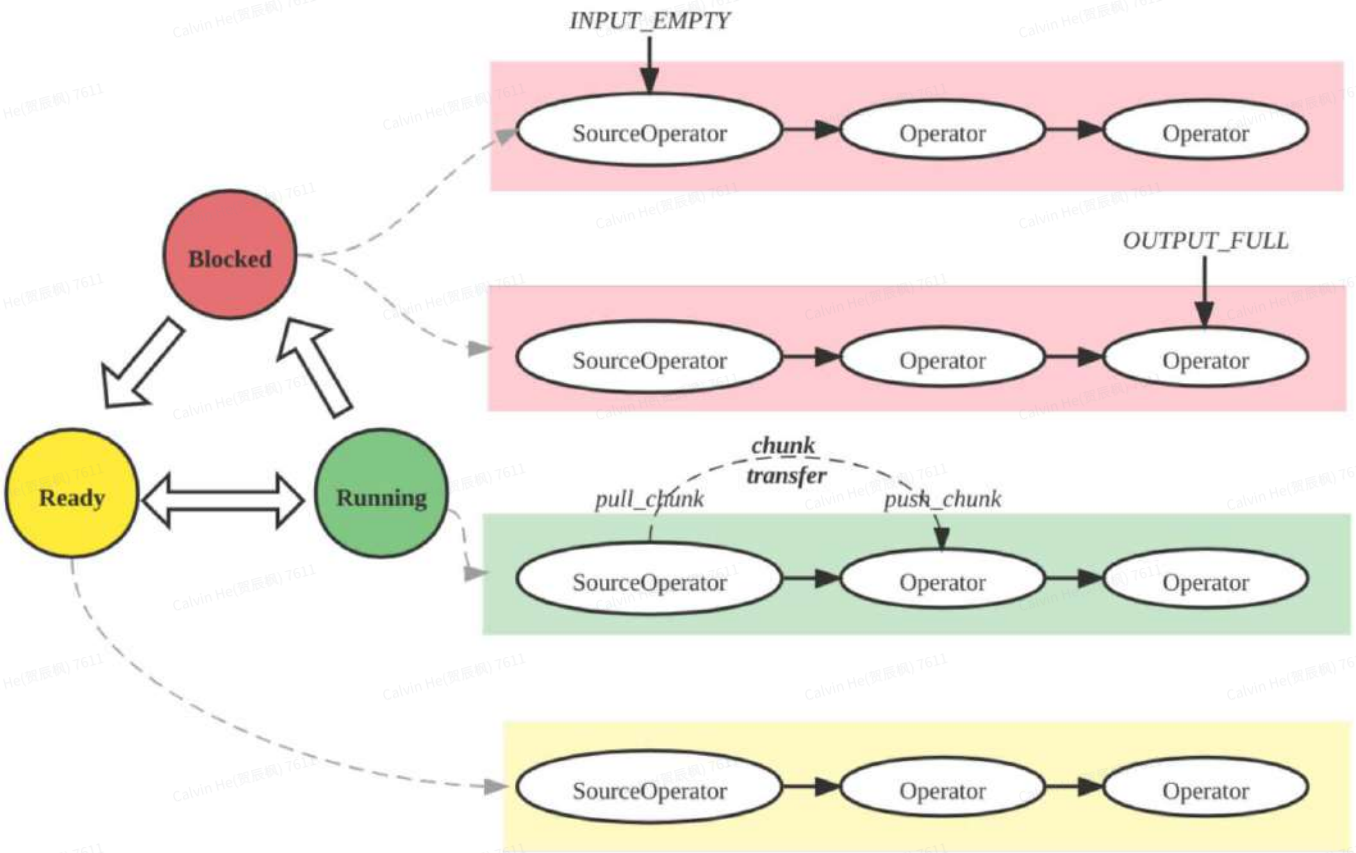
PipelineDriver的执行

PipelineDriver状态

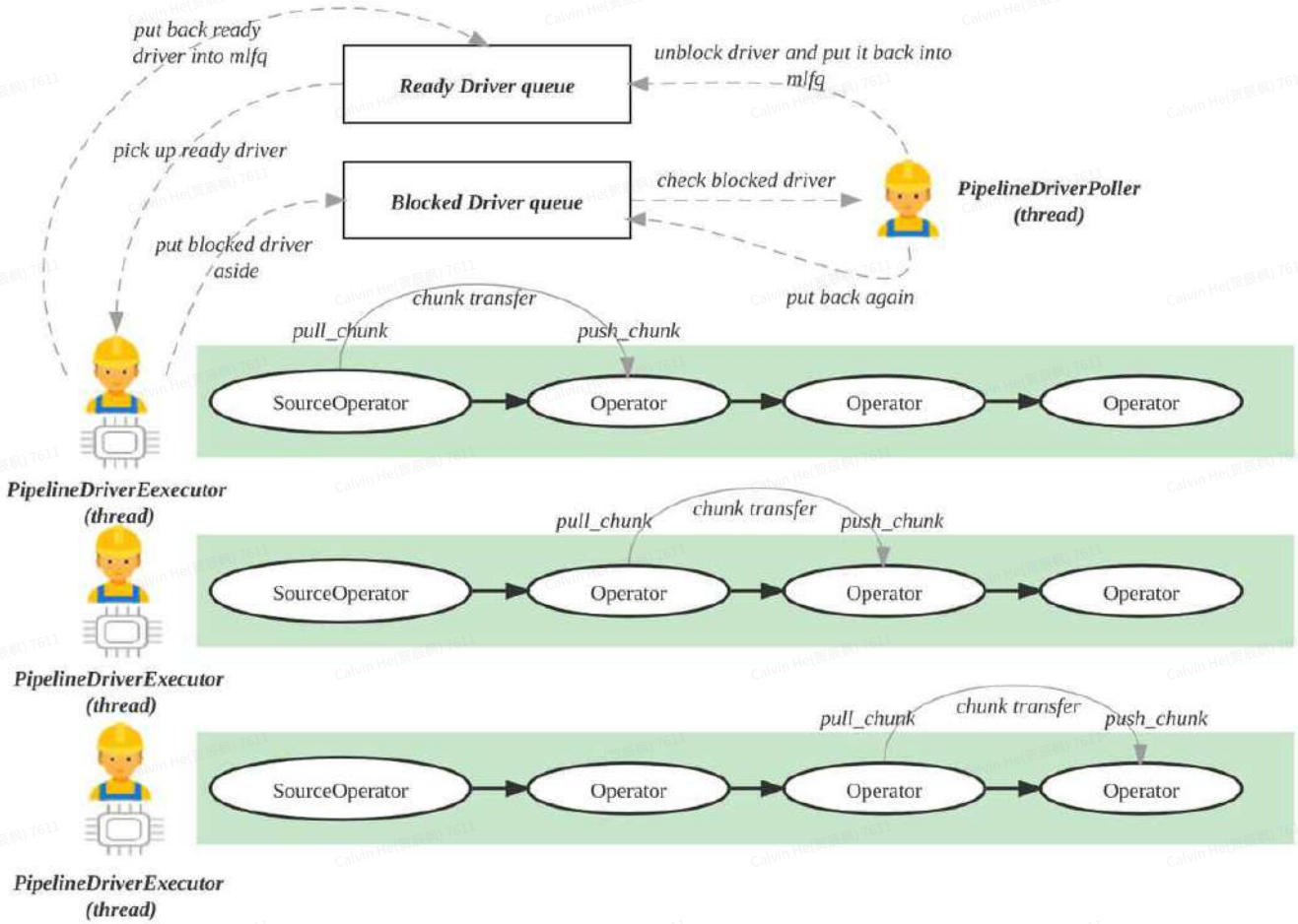
- Running
  - executed on core by a execution thread
  - PipelineDriver makes no progress is set to Blocked.
  - After a driver runs for a time slice(100ms, 100 chunks transfered), it is still not blocked, the driver is set to Ready and yield CPU to another Ready driver.
- Ready
  - Ready drivers are off-core and wait for scheduling in the ready queue.
  - Blocked drivers are set to Ready after they are unblocked.

- Execution threads pick up Ready drivers to execute and set their state to Running.

- Blocked
  - no chunk can be transfered in any two adjacent operators.
  - either source operator is INPUT_EMPTY or sink operator is OUTPUT_FULL



PipelineDriver executor/poller



开发注意事项

代码粗读

重要概念FragmentExecutor, QueryContext, FragmentContext, Pipeline, PipelineDriver, OperatorFactory和Operator等, 在 StarRocks Pipeline 执行框架 有比较详细的解释, 实际开发过程中,会涉及到一些初始化的操作, 比如:

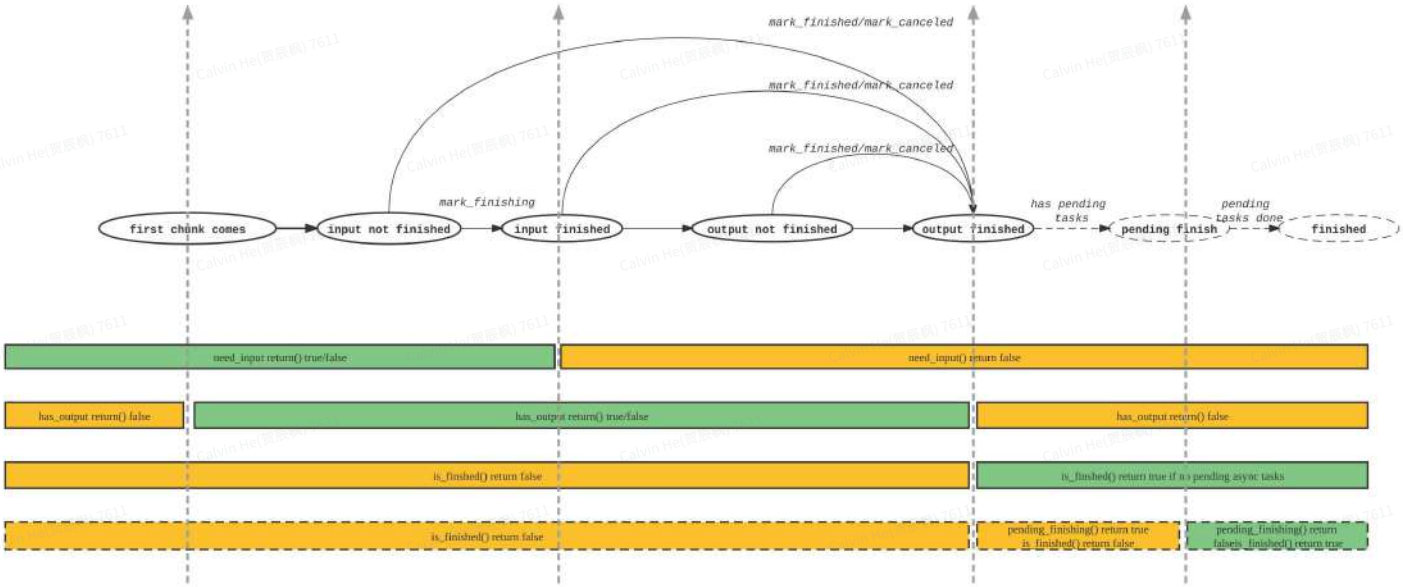
1. FragmentExecutor::prepare函数: 初始化QueryContext, FragmentContext对象, 构建由ExecNode组成的执行树, 拆分Pipeline, 拆Morsel, 实例化PipelineDriver.
2. QueryContext/FragmentContext: 用于query生命周期和Fragment Instance生命周期管理. 有些重要的query-scoped和instance-scoped对象分别放在QueryContext和FragmentContext中.
3. Pipeline: 由OperatorFactory构成, Pipeline::prepare调用OperatorFactory::prepare函数, 有些对象如果是跨PipelineDriver的并且可以只保留一份, 则可以放在OperatorFactory中, 比如conjuncts, row_descriptor.

4. PipelineDriver: 由Operator构成, PipelineDriver::prepare函数比较重要, 会调用Operator::prepare函数, 完成每个算子的初始化.

Operator的状态转化

代码be/src/exec/pipeline/operator.h中Operator的接口和状态做了比较详细的解释.

Operator的状态如下图所示:



- set_finishing表示关闭输入流;
- set_finished表示正常情况下关闭输入流和输出流;
- set_cancel同set_finished, 表示异常情况下, 关闭输入流和输出流.
- need_input()返回true, 表示算子可以接收输入,即调用push_chunk;
- has_output()返回true, 表示算子对外输出数据, 即调用pull_chunk;
- is_finished()返回true, 表示算子已经计算完毕;
- pending_finish()返回true, 表示算子依然有pending的异步任务, 尚未结束.
- 当算子未被set_finishing之前, 输入流未结束, 算子的need_input可返回true/false; set_finishing关闭输入流, 算子不再接受输入, need_input()之后一直返回false。
- 在算子的第一个输入chunk未到达之前, 算子的has_output()函数始终返回false; 当算子开始接收数据, 直到收到所有数据, 并且内部缓存的结果全部输出完成的期间, 输出流未结束, has_output()返回true/false; 之后, 输出流结束, has_output始终返回false.

- 当算子输入流和输出流都结束, 并且没有pending异步任务, 则has_output(), need_input(), pending_finish()都返回false, 而is_finished()返回true; 如果存在异步任务, 则has_output(), need_input(), is_finished()返回false, 并且pending_finish()返回true, 直到pending异步任务全部结束, pending_finish()返回false, is_finished()返回true。
- 取消计算和和后置算子提前完成, 会使当前算子也提前完成. 此时需要调用set_finished()或者set_cancel函数使算子的输入流和输出流一并结束.

推荐的Pipeline框架的扩展方式

目前, pipeline框架性的修改, 主要还是由负责执行引擎开发的同学专门完成; 其他组的同学注意是涉及到pipeline时, 做一些扩展, 比如:

1. 增加算子, 主要以Source/SinkOperator算子为主.
2. 添加一些特殊的控制逻辑, 比如导入时动态创建partition.

建议通过两种方式扩展pipeline的功能:

1. 增加新的算子
    a. 原来的非Pipeline算子改写为Pipeline算子, 比如:
        i. FileScanNode->FileScanOperator,
        ii. OlapTableSink->OlapTableSinkOperator.
    b. 增加特殊数据处理的算子: 比如:
        i. ChunkAccumulatorOperator: 小chunk攒成大chunk,
        ii. LocalExchangeOperator: 对数据流进行scatter/gather/shuffle
    c. 增加控制流相关的算子, 算子不产生数据或者以数据以passthrough方式流经算子:
        i. OlapScanPrepareOperator, NoopSinkOperator: 在ScanOperator算子执行前, 完成一些初始化操作.
        ii. CacheOperator: 计算结果passthrough给下游算子, 根据结果大小和行数, 确定是否更新缓存.
        iii. 导入时动态创建Partition的算子: 上游算子的输出直传给流经该算子, 直传给下游算子, 检查数据对应的partition是否存在, 不存在时, 动态创建partition.
    d. 利用已有的算子组合出复合算子
        i. ConjugateOperator: 将ExecNode拆出的一对Sink/Operator算子, 再次拼接成一个算子, 比如拼接AggregateBlockingSinkOperator和AggregateBlockingSourceOperator.
        ii. MultilaneOperator: 将普通的做Transform计算的算子转换为多道算子, 支持Per-Tablet计算.
2. 扩展PipelineDriver: 派生PipelineDriver类, 重写process函数. 目前还没有用到.

算子的并行化和异步化

并行化

在开发算子的过程中, 前期我们通常:

- 先实现算子的基本功能, 不考虑Pipeline并行;
- 设置pipeline_dop=1, 利用Fragment Instance并行加速.

但是当算子自身计算比较耗时, 需要scale并行加速, 则建议功能正式上线前, 实现算子的并行化. 因为:

- 输入数据在Fragment Instance间划分, 是由FE决定的, BE难以动态调整; 而Pipeline并行, 理论上是可以调整的, 输入数据和PipelineDriver的映射关系, 解决数据倾斜问题.

算子并行化可以参考:

OlapScanOperator

ResultSinkOperator

ExchangeSinkOperator

ExchangeSourceOperator

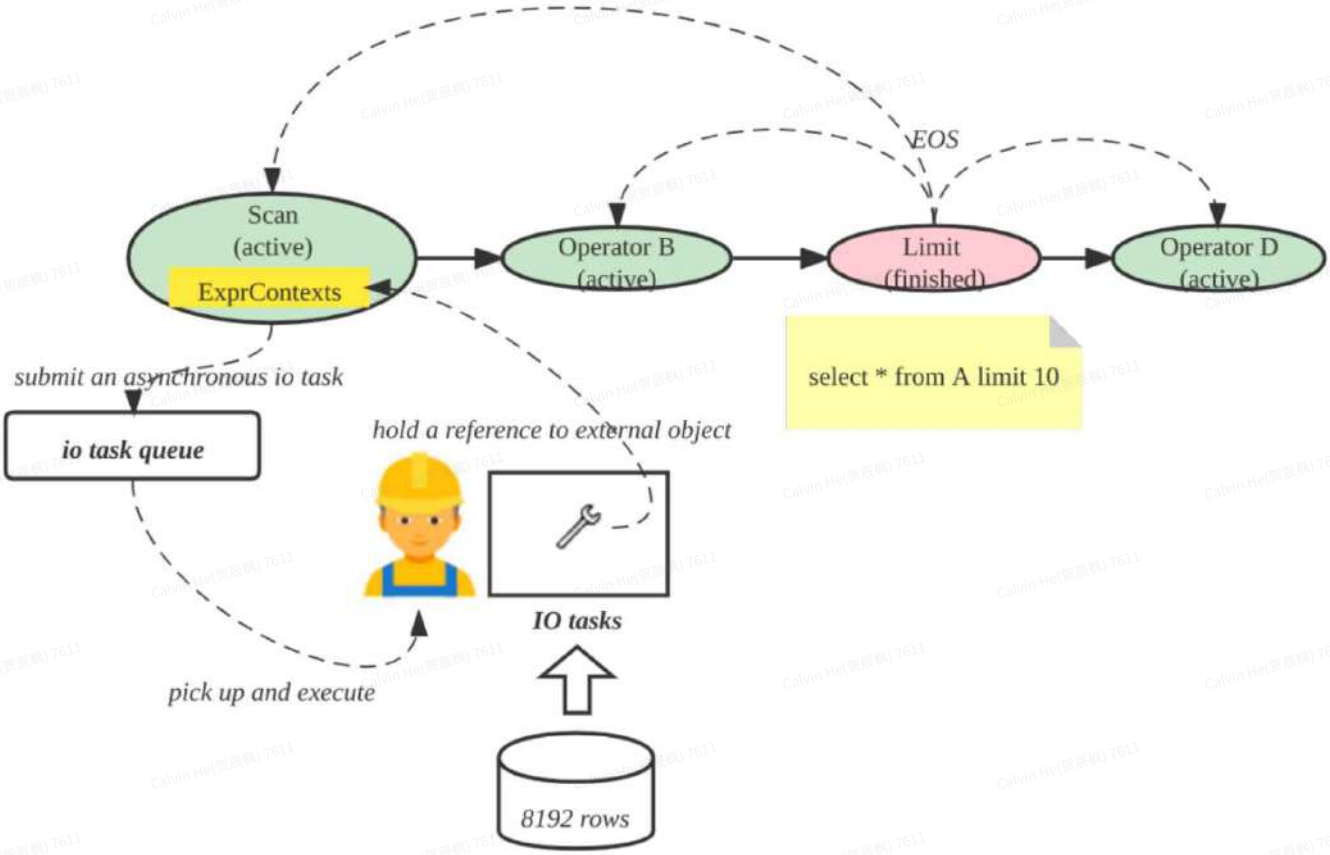
异步化

当算子有比较耗时的阻塞操作时, 必须实现异步化, 避免挂起Pipeline执行线程.

可以通过下面方式实现异步化:

- 利用已有异步接口(比如ExchangeSinkOperator使用brpc)实现异步化;
- 开辟一个线程池专门处理IO任务(比如OlapScanOperator, ConnectorScanOperator)

当PipelineDriver已经提前完成计算, 需要等待pending task完成后, 做清理工作. 如下图所示, Limit算子收到足够多的数据后, 整个Pipeline可以提前结束, 但是Scan算子提交IO任务还处于io task queue中, 如果Driver结束并且释放ScanOperator, 过了一会儿后, IO线程捡起IO任务计算, 访问ScanOperator中数据结构, 可能会触发segmentation fault.



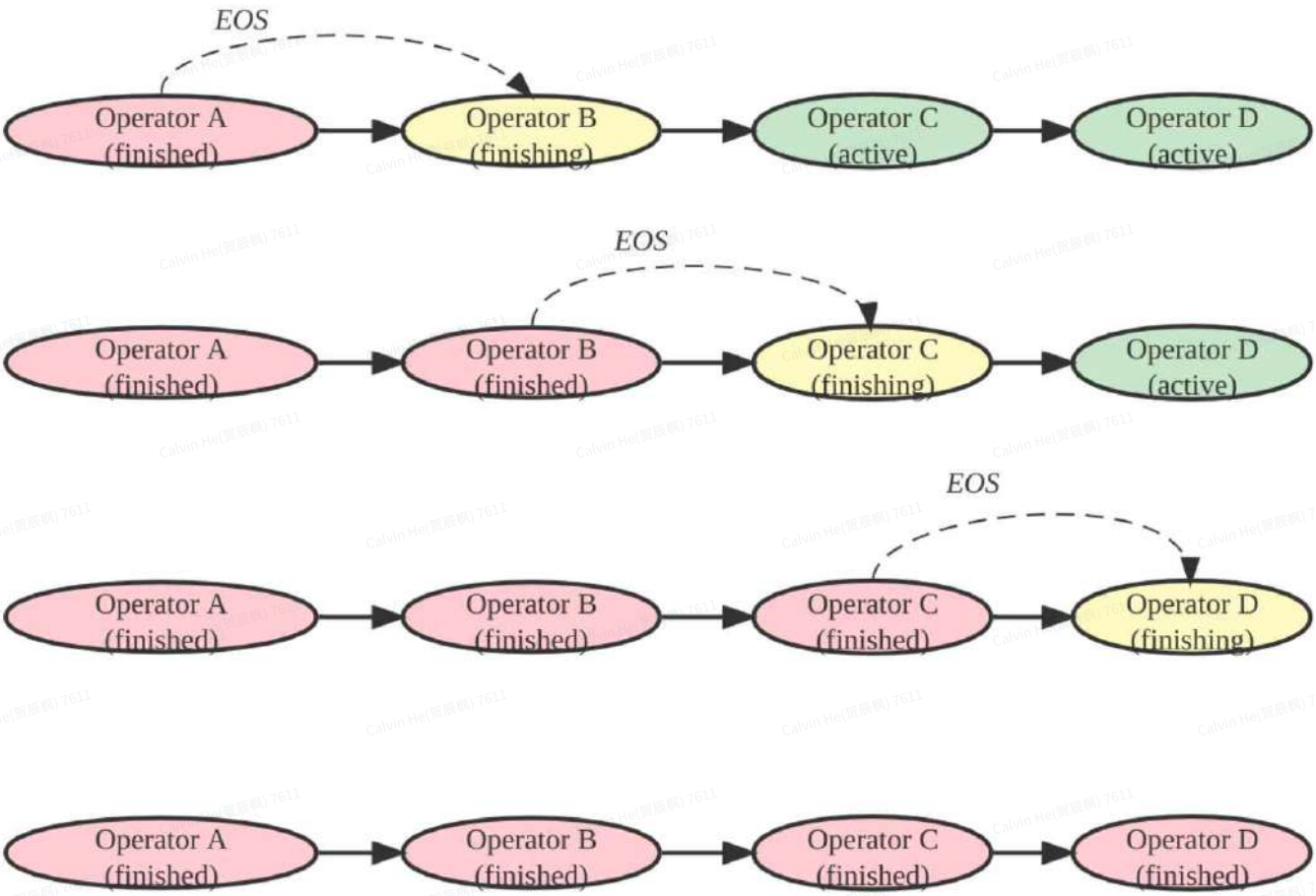
怎么解决呢？ 两种思路:

1. 使用weak_ptr: 异步IO任务使用weak_ptr引用资源, IO任务被执行时, 先对weak_ptr做lock操作获取shared_ptr, 后访问资源. 如果lock失败, 则说明, 资源已经释放, IO任务直接结束即可.
2. 算子定义pending_finishing状态. 当算子满足下面三个条件, 则pending_finishing应该返回true:
   a. 输入流已经结束: 标志为set_finishing/set_finished已经调用;
   b. 内部缓存结果已经全部输出: 标志为has_output()返回false;
   c. 有pending的异步任务尚未结束.

按照上述pending finishing语意定义pending_finish函数, pipeline执行框架中的poller线程会轮询driver的是否由pending_finish进入真正的finished()状态, 条件满足后, 会触发后续的清理工作.

Short-circuit

按照一般正常的PipelineDriver计算流程, 通常是前面的算子先于后面的算子完成.



但是存在下面几种情况, PipelineDriver中, 后面的算子先于前面的算子完成. 我们把这种现象称之为short-circuit. 造成短路的情况有:

1. Limit算子
2. HashJoin算子:
    a. 右表为空且join类型非{full outer, left outer, left anti}
    b. 右表较小, 全部probe命中, 并且join类型为{right semi, right anti} (尚未支持)
3. NestedLoopJoin算子:
    a. 右表为空

在一个Fragment Instance内部, 上游的PipelineDriver的SinkOperator, 需要把数据输出给下游的PipelineDriver的SourceOperator, 这组SinkOperator/SourceOperator之间通过Context共享数据.

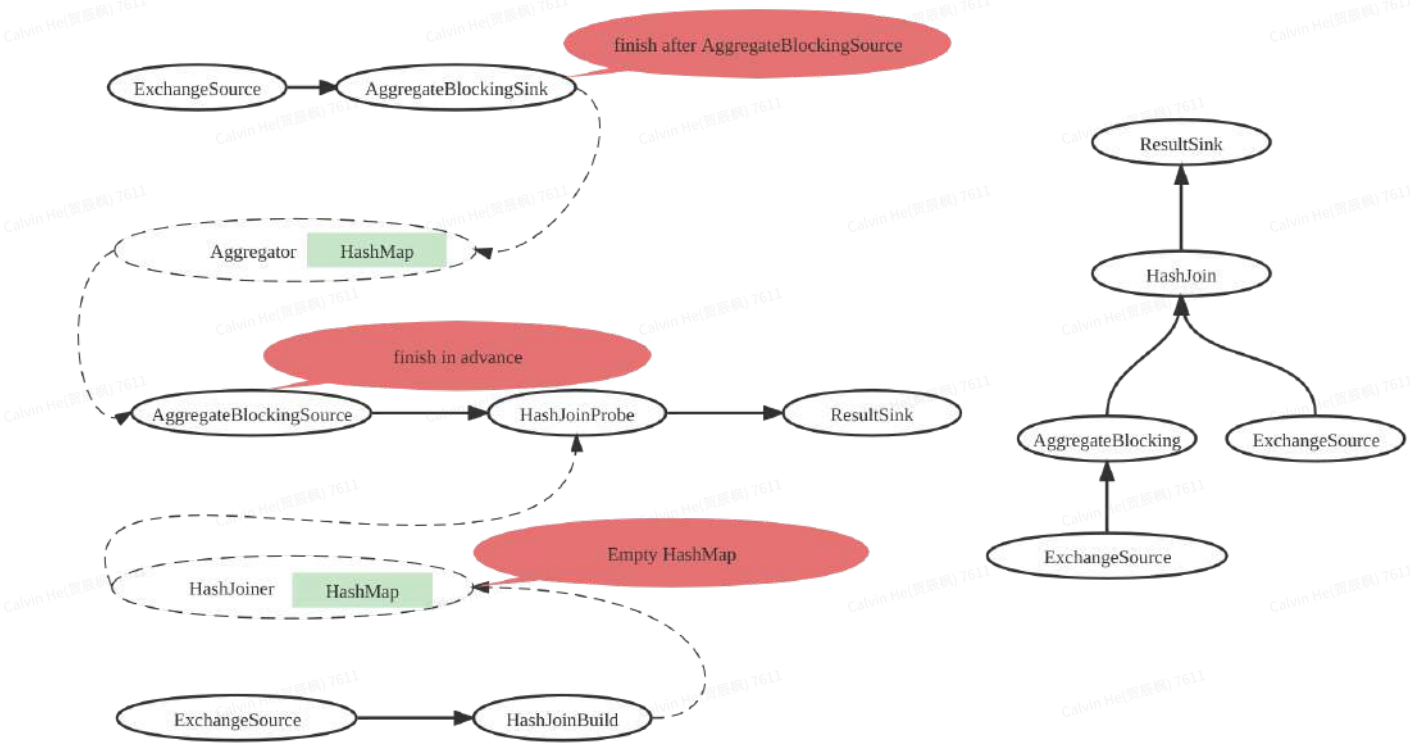
```
^ContextWithDependency$
└── ContextWithDependency      [vim be/src/exec/pipeline/context_with_dependency.h +32]
    ├── Analytor       [vim be/src/exec/vectorized/analytor.h +81]
    ├── ExceptContext  [vim be/src/exec/pipeline/set/except_context.h +41]
    ├── HashJoiner     [vim be/src/exec/vectorized/hash_joiner.h +115]
    ├── IntersectContext       [vim be/src/exec/pipeline/set/intersect_context.h +41]
    ├── NLJoinContext  [vim be/src/exec/pipeline/nljoin/nljoin_context.h +43]
    ├── OlapScanContext        [vim be/src/exec/pipeline/scan/olap_scan_context.h +46]
    ├── SortContext    [vim be/src/exec/pipeline/sort/sort_context.h +37]
    └── Aggregator     [vim be/src/exec/vectorized/aggregator.h +156]
        └── SortedStreamingAggregator [vim be/src/exec/vectorized/sorted_streaming_aggregator.h +25]
```

```
^LocalExchanger$
└── LocalExchanger     [vim be/src/exec/pipeline/exchange/local_exchange.h +33]
    ├── BroadcastExchanger     [vim be/src/exec/pipeline/exchange/local_exchange.h +132]
    ├── PartitionExchanger     [vim be/src/exec/pipeline/exchange/local_exchange.h +77]
    └── PassthroughExchanger   [vim be/src/exec/pipeline/exchange/local_exchange.h +142]
```

需要保证下面约束:

1. 上游下游SourceOperator提前结束逆向触发上游SinkOperator的提前结束.
2. 资源安全释放.
3. 需要正确实现SourceOperator::set_finished函数: 该函数一旦调用, SinkOperator的has_output/need_input需要返回false, 而is_finished() 和pending_finish()两者择一为true.

举个例子



- ExchangeSource->HashJoinBuild输入为空, HashJoiner中构建的HashMap为空; 导致HashJoinProbe提前结束.

- AggregateBlockingSource->HashJoinProbe->ResultSink中, HashJoinProbe结束, 导致AggregateBlockingSource结束. 假设此刻Aggregator中构建HashMap已经完成了一半.
- ExchangeSource->AggregateBlockingSink中AggregateBlockingSink感知到AggregateBlockingSource结束而后结束.

多路或者多输出算子的拆分

多路算子的拆分, 不详细展开了, 有兴趣的同学, 可以参考:

- 多路输入算子参考 集合算子(Union/Except/Intersect)拆分: Union、Except、Intersect 算子拆分
- 多路输出算子参考 MultiCastDataStreamSink拆分: CTE BE/Pipeline 设计文档

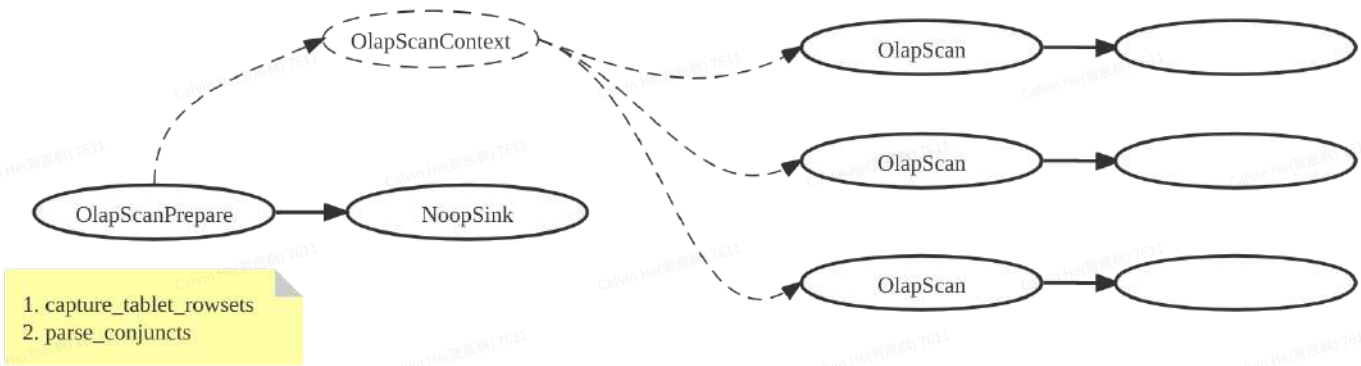
控制流

Pipeline通常可以看成是数据流的组织方式, 上游算子产生数据驱动下游算子的计算. 但事实上, 我们有时候, 不一定需要产生数据, 需要保证两个计算的同步语意: 比如A计算是B计算的前置依赖, 只有A完成后, B才能开始计算.

这种控制流的逻辑, pipeline其实也可以实现.

举个例子:

OlapScanPrepareOperator->NoopSinkOperator



同一个Pipeline实例化产生的多个PipelineDriver中, 原OlapScanOperator::preprare函数中捕获rowsets和parse_conjuncts的操作, 是相同的, 占用PipelineDriver::preprare的时间. 而且没有必要重复地prepare. 怎么只计算一次呢?

1. 将相同的计算, 提升到OlapScanOperatorFactory中, 无法处理RuntimeFilter.
2. 采用once_flag等机制初始化一次. 逻辑条理不清楚.

添加OlapScanPrepareOperator算子后, 逻辑比较清晰自然.

- 原来OlapScanNode::prepare中重复的操作挪动到OlapScanPrepareOperator中;

- OlapScanPrepareOperator和OlapScanOperator之间通过OlapScanContext同步数据;
- NoopSink是结构性的SinkOperator, 用来保证一个Pipeline有SinkOperator, 实际上并不干活.

其他方面

Pipeline执行引擎涉及的细节东西比较多,不再赘述, 大家可以参考专门的设计文档, 比如:

- Resource group资源隔离;
- ScanOperator的优化: tablet内并行, shared-scan等等;
- RuntimeFilter;
- Exchange的优化;
- 低基数词典优化;
- 复杂算子的优化;
- ColumnPool优化;
- 提升Scale能力的优化;
- Query Cache;
- Sort Aggregation;
- IO scheduler;
- MV实现;
- Profile查看;
- ...

Q&A

谢谢大家

涉及文档:

单机调度引擎的设计方案 v1

StarRocks Pipeline 执行框架